# Java Object-Oriented Programming (OOP) - Complete Guide

### 1. Encapsulation

**Definition**: Encapsulation is the bundling of data (variables) and methods that operate on that data into a single unit (class), while restricting direct access to some of the object's components.

### **Key Principles:**

- Data Hiding: Private variables cannot be accessed directly from outside the class
- Controlled Access: Public methods (getters/setters) provide controlled access to private data
- Protection: Prevents unauthorized access and modification of data

#### **Access Modifiers:**

- (private): Accessible only within the same class
- (protected): Accessible within the same package and subclasses
- (public): Accessible from anywhere
- (default) (no modifier): Accessible within the same package

Exan	nple:		
java			

```
public class BankAccount {
  private double balance; // Private data - encapsulated
  private String accountNumber; // Private data - encapsulated
 // Constructor
  public BankAccount(String accountNumber, double initialBalance) {
    this.accountNumber = accountNumber:
    this.balance = initialBalance:
  // Getter method - controlled access
  public double getBalance() {
    return balance;
  // Setter method with validation - controlled access
  public void deposit(double amount) {
    if (amount > 0) {
      balance += amount:
    } else {
      throw new IllegalArgumentException("Amount must be positive");
  public void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
      balance -= amount;
    } else {
      throw new IllegalArgumentException("Invalid withdrawal amount");
 // Private helper method - internal use only
  private boolean isValidTransaction(double amount) {
    return amount > 0;
```

- Security: Data is protected from unauthorized access
- Maintainability: Internal implementation can be changed without affecting external code
- Flexibility: Validation and business logic can be added to setter methods
- Code Organization: Related data and methods are grouped together

## 2. Inheritance

**Definition**: Inheritance is a mechanism where a new class (child/subclass) acquires properties and behaviors of an existing class (parent/superclass).

## Types of Inheritance in Java:

- Single Inheritance: One class extends another class
- Multilevel Inheritance: Class A → Class B → Class C
- Hierarchical Inheritance: Multiple classes extend from one parent class

### **Keywords:**

- (extends): Used to inherit from a class
- (super): Refers to the parent class

ample:			
ava			

```
// Parent class (Superclass)
public class Vehicle {
  protected String brand;
  protected String model;
  protected int year;
  public Vehicle(String brand, String model, int year) {
     this.brand = brand:
     this.model = model;
     this.year = year;
  public void start() {
     System.out.println("Vehicle is starting...");
  public void stop() {
     System.out.println("Vehicle is stopping...");
  public void displayInfo() {
     System.out.println("Brand: " + brand + ", Model: " + model + ", Year: " + year);
// Child class (Subclass)
public class Car extends Vehicle {
  private int numberOfDoors;
  private String fuelType;
  public Car(String brand, String model, int year, int numberOfDoors, String fuelType) {
     super(brand, model, year); // Call parent constructor
     this.numberOfDoors = numberOfDoors;
     this.fuelType = fuelType;
  // Method overriding
  @Override
  public void start() {
     System.out.println("Car engine is starting with a key...");
  // New method specific to Car
  public void honk() {
     System.out.println("Car is honking: Beep! Beep!");
```

```
// Overriding displayInfo to include car-specific information
  @Override
  public void displayInfo() {
     super.displayInfo(); // Call parent method
     System.out.println("Doors: " + numberOfDoors + ", Fuel: " + fuelType);
// Another child class
public class Motorcycle extends Vehicle {
  private boolean hasSidecar;
  public Motorcycle(String brand, String model, int year, boolean hasSidecar) {
     super(brand, model, year);
     this.hasSidecar = hasSidecar:
  @Override
  public void start() {
     System.out.println("Motorcycle is starting with a kick/button...");
  public void wheelie() {
     System.out.println("Motorcycle is doing a wheelie!");
```

- Code Reusability: Common functionality is written once in the parent class
- Method Overriding: Child classes can provide specific implementations
- **Hierarchical Organization**: Creates a natural class hierarchy
- Extensibility: New functionality can be added to child classes

### 3. Abstraction

**Definition**: Abstraction is the process of hiding complex implementation details while showing only essential features of an object.

## Types of Abstraction:

1. Abstract Classes: Cannot be instantiated, may contain abstract and concrete methods

2. <b>Interfaces</b> : Contract that classes must follow, contains only abstract methods (Java 8+ allows default methods)					
Abstract Classes:					
java					

```
// Abstract class
public abstract class Shape {
  protected String color;
  public Shape(String color) {
     this.color = color;
  // Abstract method - must be implemented by subclasses
  public abstract double calculateArea();
  public abstract double calculatePerimeter();
  // Concrete method - can be used by subclasses
  public void displayColor() {
     System.out.println("Color: " + color);
  // Another concrete method
  public final void printShapeInfo() {
     System.out.println("This is a " + color + " shape");
     System.out.println("Area: " + calculateArea());
     System.out.println("Perimeter: " + calculatePerimeter());
// Concrete class extending abstract class
public class Circle extends Shape {
  private double radius;
  public Circle(String color, double radius) {
     super(color);
     this.radius = radius:
  @Override
  public double calculateArea() {
     return Math.PI * radius * radius;
  @Override
  public double calculatePerimeter() {
     return 2 * Math.PI * radius;
public class Rectangle extends Shape {
```

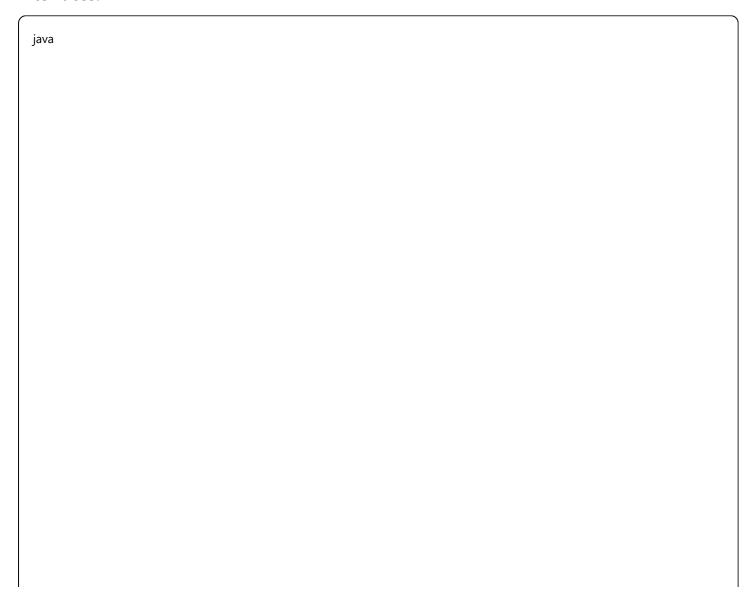
```
private double length;
private double width;

public Rectangle(String color, double length, double width) {
    super(color);
    this.length = length;
    this.width = width;
}

@Override
public double calculateArea() {
    return length * width;
}

@Override
public double calculatePerimeter() {
    return 2 * (length + width);
}
```

### Interfaces:



```
// Interface
public interface Drawable {
  // Abstract method (implicitly public and abstract)
  void draw();
  void resize(double factor):
  // Default method (Java 8+)
  default void display() {
     System.out.println("Displaying the drawable object");
  // Static method (Java 8+)
  static void printlnfo() {
     System.out.println("This is a drawable interface");
  }
  // Constant (implicitly public, static, final)
  String TYPE = "DRAWABLE";
// Another interface
public interface Colorable {
  void setColor(String color);
  String getColor();
// Class implementing multiple interfaces
public class Canvas implements Drawable, Colorable {
  private String color;
  private double size;
  public Canvas(String color, double size) {
     this.color = color;
     this.size = size:
  @Override
  public void draw() {
     System.out.println("Drawing on " + color + " canvas");
  @Override
  public void resize(double factor) {
     size *= factor;
     System.out.println("Canvas resized to: " + size);
```

```
@Override
public void setColor(String color) {
    this.color = color;
}

@Override
public String getColor() {
    return color;
}
```

- Simplification: Complex systems are represented in simple terms
- Focus on What, Not How: Users interact with objects without knowing internal details
- Contract Definition: Interfaces define what methods a class must implement
- Multiple Inheritance: Classes can implement multiple interfaces

### 4. Polymorphism

**Definition**: Polymorphism allows objects of different types to be treated as instances of the same type through a common interface, enabling one interface to represent different underlying forms (data types).

## Types of Polymorphism:

- 1. Compile-time Polymorphism (Static):
  - Method Overloading
  - Operator Overloading (limited in Java)
- 2. Runtime Polymorphism (Dynamic):
  - Method Overriding
  - Interface Implementation

### Method Overloading (Compile-time):

	ethod Overloading (Compile time).					
java						

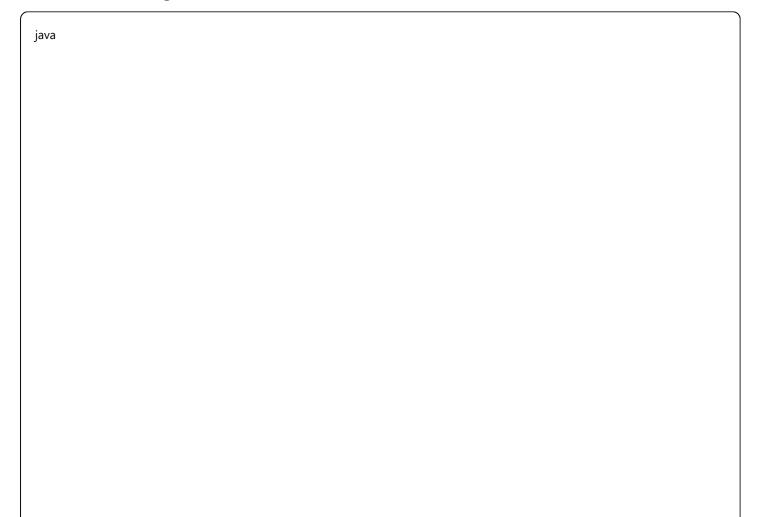
```
public class Calculator {
    // Method with 2 integer parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Method with 3 integer parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    public double add(double a, double b) {
        return a + b;
    }

    // Method with different parameter types
    public String add(String a, String b) {
        return a + b;
    }
}
```

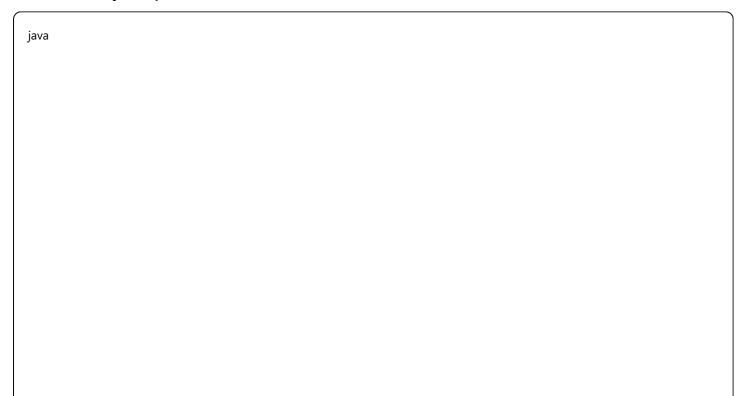
## Method Overriding (Runtime):



```
// Base class
public class Animal {
  public void makeSound() {
    System.out.println("Animal makes a sound");
  public void eat() {
    System.out.println("Animal is eating");
// Derived classes
public class Dog extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Dog barks: Woof! Woof!");
  @Override
  public void eat() {
    System.out.println("Dog is eating dog food");
  // Dog-specific method
  public void wagTail() {
    System.out.println("Dog is wagging tail");
public class Cat extends Animal {
  @Override
  public void makeSound() {
    System.out.println("Cat meows: Meow! Meow!");
  @Override
  public void eat() {
    System.out.println("Cat is eating cat food");
  // Cat-specific method
  public void purr() {
    System.out.println("Cat is purring");
```

```
// Demonstration of runtime polymorphism
public class PolymorphismDemo {
  public static void main(String[] args) {
    // Polymorphic array
     Animal[] animals = {
       new Dog(),
       new Cat(),
       new Dog(),
       new Cat()
    };
     // Runtime polymorphism in action
     for (Animal animal: animals) {
       animal.makeSound(); // Calls appropriate overridden method
       animal.eat();
                      // Calls appropriate overridden method
       // Type checking and casting
       if (animal instanceof Dog) {
         Dog dog = (Dog) animal;
         dog.wagTail();
       } else if (animal instanceof Cat) {
         Cat cat = (Cat) animal;
         cat.purr();
```

## **Interface Polymorphism:**



```
public interface PaymentProcessor {
  void processPayment(double amount);
  boolean validatePayment(double amount);
public class CreditCardProcessor implements PaymentProcessor {
  @Override
  public void processPayment(double amount) {
    System.out.println("Processing credit card payment: $" + amount);
  @Override
  public boolean validatePayment(double amount) {
    return amount > 0 && amount <= 10000:
public class PayPalProcessor implements PaymentProcessor {
  @Override
  public void processPayment(double amount) {
    System.out.println("Processing PayPal payment: $" + amount);
  @Override
  public boolean validatePayment(double amount) {
    return amount > 0 && amount <= 5000;
// Using polymorphism
public class PaymentService {
  public void makePayment(PaymentProcessor processor, double amount) {
    if (processor.validatePayment(amount)) {
       processor.processPayment(amount);
    } else {
       System.out.println("Payment validation failed");
```

- Flexibility: Same code can work with different object types
- Extensibility: New types can be added without changing existing code
- Code Reusability: Same methods can handle different object types

• Maintainability: Changes to specific implementations don't affect client code

## 5. Anonymous Inner Classes

**Definition**: Anonymous inner classes are inner classes without a name. They are used to create a one-time use class that either extends a class or implements an interface.

#### **Characteristics:**

- No explicit name
- Defined and instantiated in a single expression
- Can access final or effectively final variables from enclosing scope
- Commonly used for event handling and callbacks

### **Basic Syntax:**

```
java

// For interfaces
InterfaceName obj = new InterfaceName() {
    // implementation of interface methods
};

// For classes
ClassName obj = new ClassName() {
    // override methods or add new methods
};
```

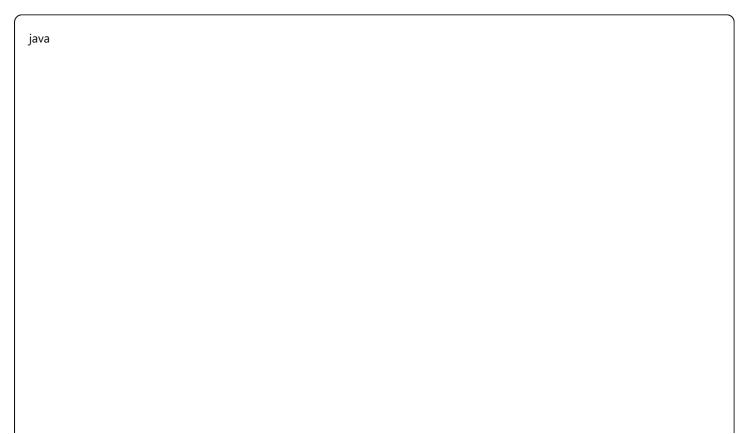
### **Examples:**

1. Implementing Interface with Anonymous Class:

java		

```
// Interface
interface Greeting {
  void greet(String name);
  void farewell(String name);
public class AnonymousExample {
  public static void main(String[] args) {
    // Anonymous class implementing interface
     Greeting greeting = new Greeting() {
       @Override
       public void greet(String name) {
          System.out.println("Hello, " + name + "!");
       @Override
       public void farewell(String name) {
          System.out.println("Goodbye, " + name + "!");
     };
     greeting.greet("Alice");
     greeting.farewell("Alice");
```

## 2. Extending Class with Anonymous Class:

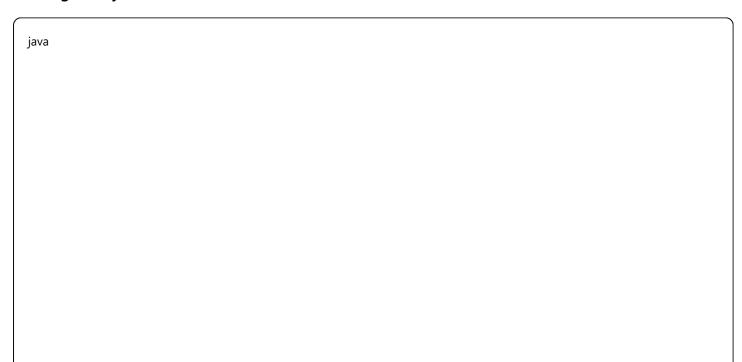


```
// Base class
abstract class Worker {
  protected String name;
  public Worker(String name) {
    this.name = name;
  public abstract void work();
  public void takeBreak() {
    System.out.println(name + " is taking a break");
public class AnonymousWorkerExample {
  public static void main(String[] args) {
    // Anonymous class extending abstract class
    Worker developer = new Worker("John") {
       @Override
       public void work() {
         System.out.println(name + " is writing code");
       // Additional method specific to this anonymous class
       public void debug() {
         System.out.println(name + " is debugging");
    };
    developer.work();
     developer.takeBreak();
    // Another anonymous class
    Worker designer = new Worker("Sarah") {
       @Override
       public void work() {
         System.out.println(name + " is creating designs");
    };
     designer.work();
     designer.takeBreak();
```

### 3. Event Handling with Anonymous Classes:

```
java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
public class ButtonExample extends JFrame {
  public ButtonExample() {
    JButton button = new JButton("Click Me");
    // Anonymous class for event handling
    button.addActionListener(new ActionListener() {
       private int clickCount = 0;
       @Override
       public void actionPerformed(ActionEvent e) {
         clickCount++;
         System.out.println("Button clicked " + clickCount + " times");
    });
    add(button);
    setSize(200, 100);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

### 4. Using Anonymous Classes with Collections:



```
import java.util.*;
public class ComparatorExample {
  public static void main(String[] args) {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
    // Anonymous class implementing Comparator
    Collections.sort(names, new Comparator<String>() {
       @Override
       public int compare(String s1, String s2) {
         // Sort by length, then alphabetically
         if (s1.length() != s2.length()) {
            return Integer.compare(s1.length(), s2.length());
         return s1.compareTo(s2);
    });
    System.out.println("Sorted names: " + names);
    // Another anonymous class for reverse sorting
    Collections.sort(names, new Comparator < String > () {
       @Override
       public int compare(String s1, String s2) {
         return s2.compareTo(s1); // Reverse order
    });
    System.out.println("Reverse sorted: " + names);
```

### 5. Anonymous Classes with Method Parameters:

java

```
interface Operation {
  int calculate(int a, int b);
class Calculator {
  public void performOperation(int a, int b, Operation op) {
     int result = op.calculate(a, b);
     System.out.println("Result: " + result);
public class OperationExample {
  public static void main(String[] args) {
     Calculator calc = new Calculator();
     // Anonymous class for addition
     calc.performOperation(10, 5, new Operation() {
       @Override
       public int calculate(int a, int b) {
          return a + b;
     });
     // Anonymous class for multiplication
     calc.performOperation(10, 5, new Operation() {
       @Override
       public int calculate(int a, int b) {
          return a * b;
     });
     // Anonymous class for power operation
     calc.performOperation(2, 3, new Operation() {
       @Override
       public int calculate(int a, int b) {
          return (int) Math.pow(a, b);
     });
```

### 6. Anonymous Classes with Local Variable Access:

java

```
public class LocalVariableExample {
  public void demonstrateVariableAccess() {
    final String prefix = "Message: ";
    int counter = 0; // Effectively final
    // Anonymous class accessing local variables
     Runnable task = new Runnable() {
       @Override
       public void run() {
         // Can access final or effectively final variables
         System.out.println(prefix + "Task is running");
         // System.out.println("Counter: " + counter); // This works too
         // Custom method in anonymous class
         printDetails();
       private void printDetails() {
         System.out.println("Task details: Anonymous runnable");
    };
    new Thread(task).start();
```

- Conciseness: Reduces code when you need a class for one-time use
- Encapsulation: Keeps implementation close to where it's used
- Event Handling: Perfect for GUI event listeners
- Callbacks: Useful for callback patterns and functional interfaces

#### **Limitations:**

- No Reusability: Cannot be reused elsewhere
- Memory Overhead: May hold references to outer class instances
- Debugging: Can be harder to debug due to lack of explicit names
- Limited Access: Can only access final or effectively final local variables

### Lambda Expressions (Modern Alternative):

Since Java 8, many use cases for anonymous classes can be replaced with lambda expressions for functional interfaces:

```
java

// Old way with anonymous class
button.addActionListener(new ActionListener() {
     @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
     }
});

// New way with lambda expression
button.addActionListener(e -> System.out.println("Button clicked"));
```

## **Summary**

These five OOP concepts work together to create robust, maintainable, and scalable Java applications:

- Encapsulation protects data and provides controlled access
- Inheritance promotes code reuse and establishes relationships
- Abstraction hides complexity and defines contracts
- Polymorphism enables flexible and extensible code
- Anonymous Inner Classes provide concise implementations for one-time use scenarios

Understanding and properly implementing these concepts is fundamental to effective Java programming and object-oriented design.